Suggested Solutions to HW#4 Problems

1. Suppose you are working on an audio processor class. The class has the following method that compresses audio files in parallel using multiple threads:

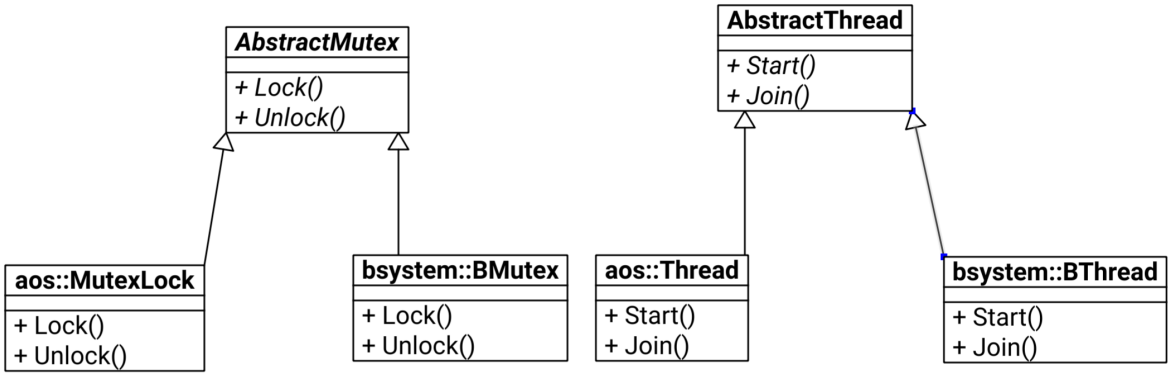
```
using namespace std;
void AudioProcessor::ParallelCompress(
    const vector<AudioFile>& files,
    int nThreads) {
  vector<unique_ptr<aos::Thread>> threads(
      nThreads);
  // Creates a aos::MutexLock instance.
  unique_ptr<aos::MutexLock> lock(
      new aos::MutexLock());

  int fIndex = files.size();
  for (int i = 0; i < nThreads; ++i) {
    // Create a aos::Thread instance.
    threads[i].reset(new aos::Thread());
    // Run the lambda on a thread.
    threads[i]->Start([&](){
      while (true) {
        // Get an unprocessed AudioFile instance.
        lock->Lock();
        if (fIndex == files.size()) {
          // Returning from the lambda will
          // terminate the thread.
          return;
        }
        const auto& file = files[fIndex++];
        lock->Unlock();

        // Compress the AudioFile instance.
        DoCompress(file);
      }
    });
  }

  // Wait until all threads are terminated.
  for (auto& thread : threads) {
    thread->Join();
  }
}
```

Now you are supporting a new platform named bsystem that provides the bsystem::BThread and bsystem::BMutex classes for manipulating threads and mutexes. Here is the class diagram of the threading classes:

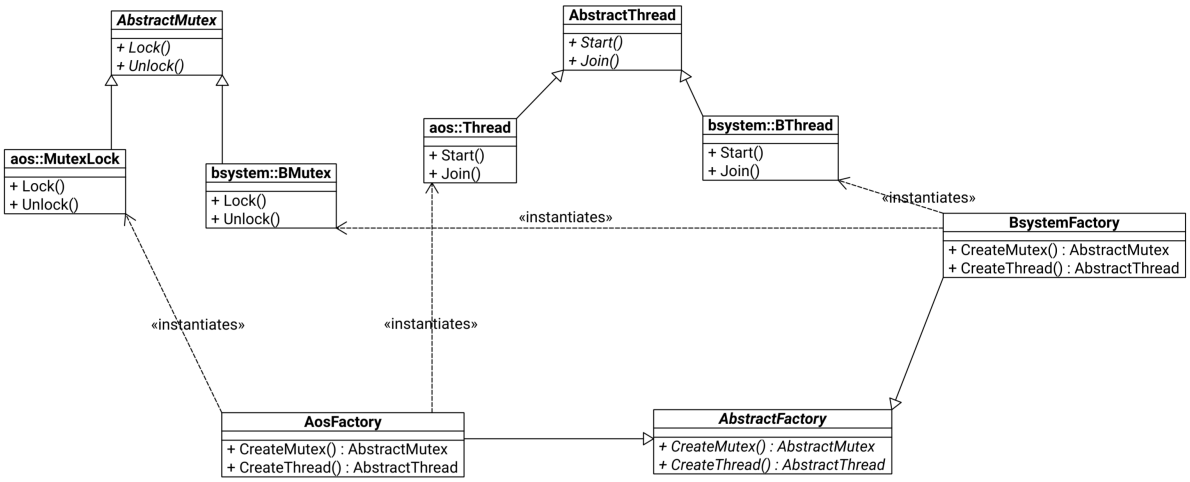


(a) (15 %) What design pattern can you use in method AudioProcessor::ParallelCompress() for instantiating the threading classes of aos and bsystem without depending on the concrete classes in the method?

Solution. The abstract factory pattern. □

(b) (25 %) Please provide the class diagram of the design and also show the reimplemented method in c++.

Solution. The class diagram:



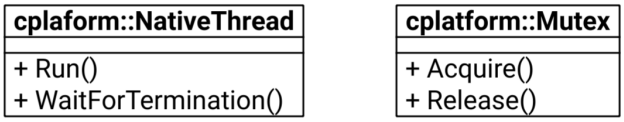
The method after applying the pattern:

```
1  using namespace std;
2  void AudioProcessor::ParallelCompress(
3      const vector<AudioFile>& files,
4      int nThreads,
5      AbstractFactory*factory) {
6    vector<unique_ptr<AbstractThread>> threads(
7        nThreads);
8    // Creates a aos::MutexLock instance.
9    unique_ptr<AbstractLock> lock(
10       factory->CreateLock());
```

```
11
12    int fIndex = files.size();
13    for (int i = 0; i < nThreads; ++i) {
14      // Create a aos::Thread instance.
15      threads[i].reset(factory->CreateThread());
16      // Run the lambda on a thread.
17      threads[i]->Start([&](){
18        while (true) {
19          // Get an unprocessed AudioFile instance.
20          lock->Lock();
21          if (fIndex == files.size()) {
22            // Returning from the lambda will
23            // terminate the thread.
24            return;
25          }
26          const auto& file = files[fIndex++];
27          lock->Unlock();
28
29          // Compress the AudioFile instance.
30          DoCompress(file);
31        }
32      });
33    }
34
35    // Wait until all threads are terminated.
36    for (auto& thread : threads) {
37      thread->Join();
38    }
39  }
```

□

2. Now you are supporting another platform, cplatform, that provides the following threading classes:



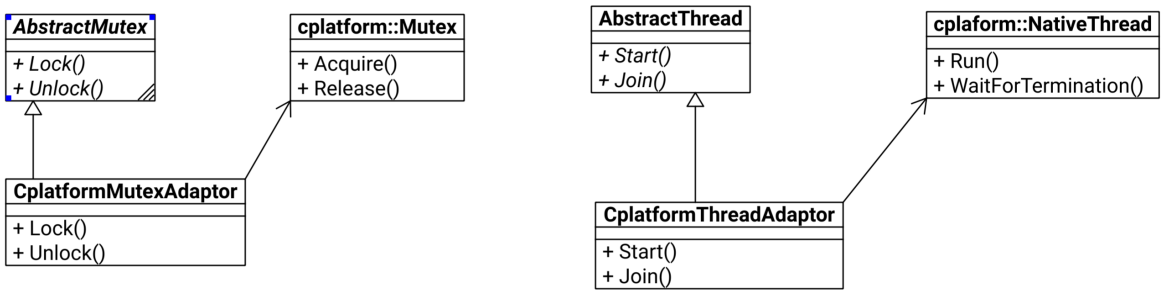
It can be seen that the cplatform threading classes provide similar functions with an incompatible interface. You need extra work to make the method support cplatform.

(a) (15 %) Which design pattern can you use to make AudioProcessor::ParallelCompress() use the cplatform classes without depending on its interface?

Solution. The adaptor pattern. □

(b) (15 %) Please provide the design in a class diagram.

Solution. The class diagram:



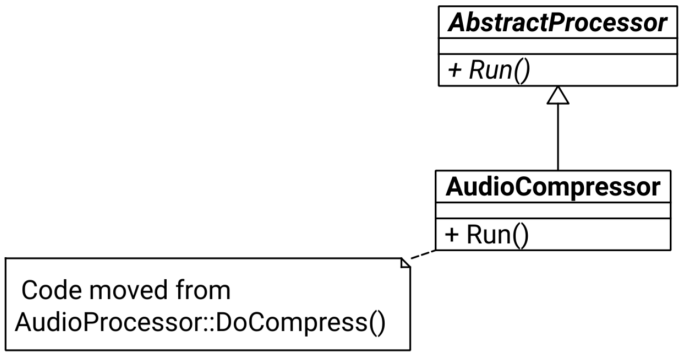
□

3. The capability of running computation-intensive workloads on separate threads is useful not only for audio compression but also for other audio processing functions like removing background noise, echo cancellation, speech recognition, etc. You renamed the method and parameterized it with an AbstractProcessor argument:

```
1  using namespace std;
2  void AudioProcessor::ParallelProcess(
3      const vector<AudioFile>& files,
4      int nThreads,
5      AbstractProcessor* processor) {
6    vector<unique_ptr<aos::Thread>> threads(
7        nThreads);
8    // Create a aos::MutexLock instance.
9    unique_ptr<aos::MutexLock> lock(
10       new aos::MutexLock());
11
12   int fIndex = files.size();
13   for (int i = 0; i < nThreads; ++i) {
14     // Create a aos::Thread instance.
15     threads[i].reset(new aos::Thread());
16
17     // Run the lambda on a thread.
18     threads[i]->Start([&](){
19       while (true) {
20         // Get an unprocessed AudioFile instance.
21         lock->Lock();
22         if (fIndex == files.size()) {
23           // Returning from the lambda will
24           // terminate the thread.
25           return;
26         }
27         const auto& file = files[fIndex++];
28         lock->Unlock();
29
30         // Run the processor with the AudioFile
31         //instance.
32         processor->Run(file);
```

```
33          }
34        });
35    }
36
37      // Wait until all threads are terminated.
38    for (auto& thread : threads) {
39      thread->Join();
40    }
41  }
```

The original AudioProcessor::DoCompress() method is moved into AudioCompress::Run():

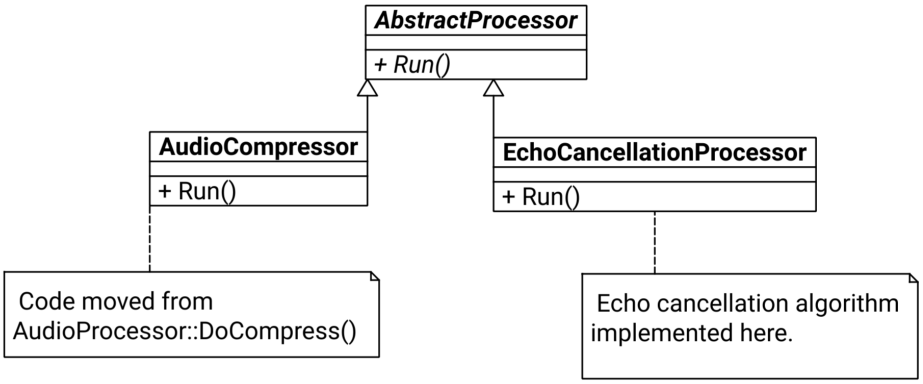


(a) (15 %) Which pattern is used to refactor the design to make it support generic processor implementations?

Solution. The command pattern. □

(b) (15 %) Please add the support of EchoCancellation in a class diagram.

Solution.



□